Beej’s Guide to Interprocess Communication

Brian “Beej Jorgensen” Hall

v1.1.4, Copyright © February 4, 2023

Chapter 4

Pipes

There is no form of IPC that is simpler than pipes. Implemented on every flavor of Unix, `pipe()` and `fork()` make up the functionality behind the "`|`" in "`ls | more`". They are marginally useful for cool things, but are a good way to learn about basic methods of IPC.

Since they're so very very easy, I shant spent much time on them. We'll just have some examples and stuff.

4.1 "These pipes are clean!"

Wait! Not so fast. I might need to define a "file descriptor" at this point. Let me put it this way: you know about "`FILE*`" from `stdio.h`, right? You know how you have all those nice functions like `fopen()`, `fclose()`, `fwrite()`, and so on? Well, those are actually high level functions that are implemented using *file descriptors*, which use system calls such as `open()`, `creat()`, `close()`, and `write()`. File descriptors are simply `int`s that are analogous to `FILE*`'s in `stdio.h`.

For example, `stdin` is file descriptor "0", `stdout` is "1", and `stderr` is "2". Likewise, any files you open using `fopen()` get their own file descriptor, although this detail is hidden from you. (This file descriptor can be retrived from the `FILE*` by using the `fileno()` macro from `stdio.h`.)

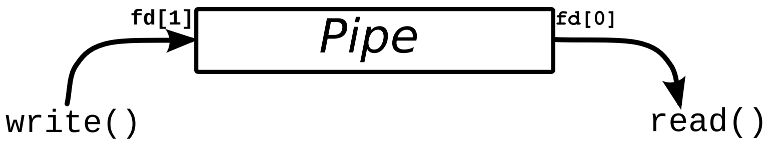

Figure 4.1: How a pipe is organized.

Basically, a call to the `pipe()` function returns a pair of file descriptors. One of these descriptors is connected to the write end of the pipe, and the other is connected to the read end. Anything can be written to the pipe, and read from the other end in the order it came in. On many systems, pipes will fill up after you write about 10K to them without reading anything out.

As a [useless example](https://beej.us/guide/bgipc/source/examples/pipe1.c], the following program creates, writes to, and reads from a pipe.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];
    char buf[30];

    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
    }

    printf("writing to file descriptor #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("reading from file descriptor #%d\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("read \"%s\"\n", buf);

    return 0;
}
```

As you can see, `pipe()` takes an array of two `int`s as an argument. Assuming no errors, it connects two file descriptors and returns them in the array. The first element of the array is the reading-end of the pipe, the second is the writing end.

4.2 `fork()` and `pipe()`—you have the power!

From the above example, it's pretty hard to see how these would even be useful. Well, since this is an IPC document, let's put a `fork()` in the mix and see what happens. Pretend that you are a top federal agent assigned to get a child process to send the word "test" to the parent. Not very glamorous, but no one ever said computer science would be the X-Files, Mulder.

First, we'll have the parent make a pipe. Secondly, we'll `fork()`. Now, the `fork()` man page tells us that the child will receive a copy of all the parent's file descriptors, and this includes a copy of the pipe's file descriptors. *Alors*, the child will be able to send stuff to the write-end of the pipe, and the parent will get it off the read-end. [flx[Like this|pipe2.c)[1], the following program creates, writes to, and reads from a pipe.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];
    char buf[30];

    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
    }

    printf("writing to file descriptor #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("reading from file descriptor #%d\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("read \"%s\"\n", buf);

    return 0;
}
```

As you can see, `pipe()` takes an array of two `int`s as an argument. Assuming no errors, it connects two

[1]https://beej.us/guide/bgipc/source/examples/pipe1.c

file descriptors and returns them in the array. The first element of the array is the reading-end of the pipe, the second is the writing end.

4.3 `fork()` and `pipe()`—you have the power!

From the above example, it's pretty hard to see how these would even be useful. Well, since this is an IPC document, let's put a `fork()` in the mix and see what happens. Pretend that you are a top federal agent assigned to get a child process to send the word "test" to the parent. Not very glamorous, but no one ever said computer science would be the X-Files, Mulder.

First, we'll have the parent make a pipe. Secondly, we'll `fork()`. Now, the `fork()` man page tells us that the child will receive a copy of all the parent's file descriptors, and this includes a copy of the pipe's file descriptors. *Alors*, the child will be able to send stuff to the write-end of the pipe, and the parent will get it off the read-end. [flx[Like this|pipe2.c]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];
    char buf[30];

    pipe(pfds);

    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }

    return 0;
}
```

Please note, your programs should have a lot more error checking than mine do. I leave it out on occasion to help keep things clear.

Anyway, this example is just like the previous one, except now we `fork()` of a new process and have it write to the pipe, while the parent reads from it. The resultant output will be something similar to the following:

```
PARENT: reading from pipe
 CHILD: writing to the pipe
 CHILD: exiting
PARENT: read "test"
```

In this case, the parent tried to read from the pipe before the child writes to it. When this happens, the parent is said to *block*, or sleep, until data arrives to be read. It seems that the parent tried to read, went to sleep, the child wrote and exited, and the parent woke up and read the data.

Hurrah!! You've just don't some interprocess communication! That was dreadfully simple, huh? I'll bet you are still thinking that there aren't many uses for `pipe()` and, well, you're probably right. The other forms of IPC are generally more useful and are often more exotic.

4.4 The search for Pipe as we know it

In an effort to make you think that pipes are actually reasonable beasts, I'll give you an example of using `pipe()` in a more familiar situation. The challenge: implement "`ls | wc -l`" in C.

This requires usage of a couple more functions you may never have heard of: `exec()` and `dup()`. The `exec()` family of functions replaces the currently running process with whichever one is passed to `exec()`. This is the function that we will use to run `ls` and `wc -l`. `dup()` takes an open file descriptor and makes a clone (a duplicate) of it. This is how we will connect the standard output of the `ls` to the standard input of `wc`. See, stdout of `ls` flows into the pipe, and the stdin of `wc` flows in from the pipe. The pipe fits right there in the middle!

Anyway, here is the code[2]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];

    pipe(pfds);

    if (!fork()) {
        close(1);       /* close normal stdout */
        dup(pfds[1]);   /* make stdout same as pfds[1] */
        close(pfds[0]); /* we don't need this */
        execlp("ls", "ls", NULL);
    } else {
        close(0);       /* close normal stdin */
        dup(pfds[0]);   /* make stdin same as pfds[0] */
        close(pfds[1]); /* we don't need this */
        execlp("wc", "wc", "-l", NULL);
    }

    return 0;
}
```

I'm going to make another note about the `close()`/`dup()` combination since it's pretty weird. `close(1)` frees up file descriptor 1 (standard output). `dup(pfds[1])` makes a copy of the write-end of the pipe in the first available file descriptor, which is "1", since we just closed that. In this way, anything that `ls` writes to standard output (file descriptor 1) will instead go to `pfds[1]` (the write end of the pipe). The `wc` section of code works the same way, except in reverse.

4.5 Summary

There aren't many of these for such a simple topic. In fact, there are nearly just about none. Probably the best use for pipes is the one you're most accustomed to: sending the standard output of one command to the standard input of another. For other uses, it's pretty limiting and there are often other IPC techniques that work better.

[2]https://beej.us/guide/bgipc/source/examples/pipe3.c

Chapter 5

FIFOs

A FIFO ("First In, First Out", pronounced "Fy-Foh") is sometimes known as a *named pipe*. That is, it's like a pipe, except that it has a name! In this case, the name is that of a file that multiple processes can `open()` and read and write to.

This latter aspect of FIFOs is designed to let them get around one of the shortcomings of normal pipes: you can't grab one end of a normal pipe that was created by an unrelated process. See, if I run two individual copies of a program, they can both call `pipe()` all they want and still not be able to speak to one another. (This is because you must `pipe()`, then `fork()` to get a child process that can communicate to the parent via the pipe.) With FIFOs, though, each unrelated process can simply `open()` the pipe and transfer data through it.

5.1 A New FIFO is Born

Since the FIFO is actually a file on disk, you have to do some fancy-schmancy stuff to create it. It's not that hard. You just have to call `mknod()` with the proper arguments. Here is a `mknod()` call that creates a FIFO:

```
mknod("myfifo", S_IFIFO | 0644 , 0);
```

In the above example, the FIFO file will be called "`myfifo`". The second argument is the creation mode, which is used to tell `mknod()` to make a FIFO (the `S_IFIFO` part of the OR) and sets access permissions to that file (octal 644, or `rw-r--r--`) which can also be set by ORing together macros from `sys/stat.h`. This permission is just like the one you'd set using the `chmod` command. Finally, a device number is passed. This is ignored when creating a FIFO, so you can put anything you want in there.

(An aside: a FIFO can also be created from the command line using the Unix `mknod` command.)

5.2 Producers and Consumers

Once the FIFO has been created, a process can start up and open it for reading or writing using the standard `open()` system call.

Since the process is easier to understand once you get some code in your belly, I'll present here two programs which will send data through a FIFO. One is `speak.c` which sends data through the FIFO, and the other is called `tick.c`, as it sucks data out of the FIFO.

Here is `speak.c`[1]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
```

[1]https://beej.us/guide/bgipc/source/examples/speak.c

```c
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("waiting for readers...\n");
    fd = open(FIFO_NAME, O_WRONLY);
    printf("got a reader--type some stuff\n");

    while (gets(s), !feof(stdin)) {
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("write");
        else
            printf("speak: wrote %d bytes\n", num);
    }

    return 0;
}
```

What `speak` does is create the FIFO, then try to `open()` it. Now, what will happen is that the `open()` call will *block* until some other process opens the other end of the pipe for reading. (There is a way around this—see `O_NDELAY`, below.) That process is `tick.c`[2], shown here:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");

    do {
        if ((num = read(fd, s, 300)) == -1)
            perror("read");
```

[2]https://beej.us/guide/bgipc/source/examples/tick.c

```
        else {
            s[num] = '\0';
            printf("tick: read %d bytes: \"%s\"\n", num, s);
        }
    } while (num > 0);

    return 0;
}
```

Like `speak.c`, `tick` will block on the `open()` if there is no one writing to the FIFO. As soon as someone opens the FIFO for writing, `tick` will spring to life.

Try it! Start `speak` and it will block until you start `tick` in another window. (Conversely, if you start `tick`, it will block until you start `speak` in another window.) Type away in the `speak` window and `tick` will suck it all up.

Now, break out of `speak`. Notice what happens: the `read()` in `tick` returns 0, signifying EOF. In this way, the reader can tell when all writers have closed their connection to the FIFO. "What?" you ask "There can be multiple writers to the same pipe?" Sure! That can be very useful, you know. Perhaps I'll show you later in the document how this can be exploited.

But for now, lets finish this topic by seeing what happens when you break out of `tick` while `speak` is running. "Broken Pipe"! What does this mean? Well, what has happened is that when all readers for a FIFO close and the writer is still open, the writer will receiver the signal SIGPIPE the next time it tries to `write()`. The default signal handler for this signal prints "Broken Pipe" and exits. Of course, you can handle this more gracefully by catching SIGPIPE through the `signal()` call.

Finally, what happens if you have multiple readers? Well, strange things happen. Sometimes one of the readers get everything. Sometimes it alternates between readers. Why do you want to have multiple readers, anyway?

5.3 `O_NDELAY`! I'm UNSTOPPABLE!

Earlier, I mentioned that you could get around the blocking `open()` call if there was no corresponding reader or writer. The way to do this is to call `open()` with the `O_NDELAY` flag set in the mode argument:

```
fd = open(FIFO_NAME, O_WRONLY | O_NDELAY);
```

This will cause `open()` to return `-1` if there are no processes that have the file open for reading.

Likewise, you can open the reader process using the `O_NDELAY` flag, but this has a different effect: all attempts to `read()` from the pipe will simply return `0` bytes read if there is no data in the pipe. (That is, the `read()` will no longer block until there is some data in the pipe.) Note that you can no longer tell if `read()` is returning `0` because there is no data in the pipe, or because the writer has exited. This is the price of power, but my suggestion is to try to stick with blocking whenever possible.

5.4 Concluding Notes

Having the name of the pipe right there on disk sure makes it easier, doesn't it? Unrelated processes can communicate via pipes! (This is an ability you will find yourself wishing for if you use normal pipes for too long.) Still, though, the functionality of pipes might not be quite what you need for your applications. Message queues might be more your speed, if your system supports them.